MATHLIB Review

Author: Walter E. Brown
Author: Marc Paterno
Revision: 1.34
Date: 2004-02-16

Table of Contents

1 Introduction

1.1 Mission and Scope of this Review

We have been asked, on behalf of CMS, to review **A Proposal for a C++ MATHLIB** by René Brun. We address such issues as *design*, *organization*, and *coverage* of the proposed MATHLIB product, and provide technical analysis and commentary regarding numerous aspects of the proposal.

1.2 Overview

The proposal presents a case for the creation of several mathematically-oriented libraries, collectively targeting the needs of the HEP community. The proposal document addresses, in its first three pages, selected issues related to the design and organization of such a project; the bulk of the remaining 21+ pages of the proposal analyze the proposed project's coverage. A brief bibliography concludes the proposal.

1.3 General Impressions

We agree that a project to produce libraries along the general lines outlined by the proposal could be a valuable contribution to the identified constituency.

Much of the proposal's nomenclature seems specific to, and oriented toward implementation via, ROOT. We believe that the target community would benefit greatly from a presentation free of assumptions regarding any selected target environment.

2 Underpinnings

2.1 Project Goals

In order to guide the design of any project, the goals for the project must be made clear. This proposal neither sets forth nor refers to a clear set of goals for a MATHLIB product. Although such goals were presumably known, we believe the proposal would be improved by formally and clearly articulating them.

> **We recommend** that requirements for the MATHLIB product be formally stated. If such goals have been provided elsewhere, those goals should be referenced in this proposal.

2.2 Why C++?

While lacking a rationale describing the *purpose* of the library, the proposal does contain "reasons to have a true C++ library." The proposal does not spell out what makes a "true" C++ library; here is the list of features we think necessary:

1. Use of *object-orientation* where appropriate,
2. Use of *genericity* (templates, if you like) where appropriate,
3. Use of *procedural* paradigm where appropriate,
4. Use of *functional* paradigm[1] where appropriate,
5. Support for clients' use of multiple paradigms where possible,
6. Support for extensibility by clients.[2]
7. Adherence to well-known design principles, *e.g.* separation of concerns and "pay for only what you use," also known as the zero-overhead principle.

2.3 Critique of Points in the Proposal

The proposal presents the following six "reasons to have a true C++ library"; we address each in turn.

> 1. We want to interact with real objects (data and algorithms), not just algorithms.

Such a sweeping statement seems too strong. For some problems, interacting with objects will be an appropriate solution -- but not in every case. As we state above (Why C++?), one of the strengths of C++ is its support of many programming paradigms. We recommend against *a priori* restrictions that may not be appropriate in all cases.

> **We recommend** the appropriate use of all the features of modern C++, in the appropriate contexts. The should be no *a priori* demand to use a single paradigm for all contexts, nor should there be an *a priori* refusal to use any paradigm.

> 2. We want to provide higher level interfaces hiding the implementation details (algorithms). A true Object-Oriented API should remain stable if internal storage or algorithms change. One can imagine the Mathlib classes being improved over time, or adapted to standard algorithms that could come with the new C++ versions.

We agree that hiding irrelevant implementation detail is important, but clearly an object interface isn't always needed. For example, few know (and even fewer care) what algorithm is used to compute `sin(x)`. Even if someone produces a better algorithm, the interface stays the same. But if, for example, a better way of reporting domain errors were produced, an interface change would be quite likely, and, under the circumstance, desirable. See a below for more on the subject of internal state for algorithms as it affects efficiency in thread-safe programs.

> 3. Many classes require a good graphics interface. A large subset of CERNLIB or GSL has to do with functions. Visualizing a function requires to know some of its properties, eg, singularities or asymptotic behaviours. This does not mean that the function classes must have built-in graphics. But they must be able to call graphics service classes able to exploit the algorithms in the functions library.

We agree with the importance of graphics in a modern environment. Assuming "they" refers to the *functions*, we strongly disagree with the rationale.

The business of graphics is very complex. There is a wide variety of graphics libraries available, to say nothing about the number of graphics devices, resolutions, domain and range choices, *etc.*, not all of which are applicable to all graphics.

There are at least three distinct functionalities under discussion:

- Computation of the value of the function at a point,
- Reporting (or computation) of features of the function, and
- Graphical display of the value of the function over some range.

The principle known as separation of concerns strongly suggests that these different tasks not be coupled. Instead, they should interact, where necessary, through a well-defined interface that permits independent replacement of any one of them.

> 4. Many objects need operators (matrices, vectors, physics vectors, *etc.*).

We agree that operator notation should be used where appropriate. However, operator notation does not require member functions. (See Exploration of Selected Alternatives, below).

5. We want to embed these objects in a data model. Users start to request that the math library takes care of memory management and/or persistence of the object. See for instance the LHC-feedback[3], where persistence of the CLHEP was requested. The user would like to save and restore random-generator seeds *etc.*

We find that this requirement is too imprecise to address in detail. Certainly many of those things represented as objects should provide for persistability (the example of the seed for a random generator is one such case). In other cases (special functions, for example), persistability does not seem to apply.

In those cases where persistability is important, it should be a goal of the MATHLIB product to allow for persistence in a manner to be chosen by the user. The forthcoming **Technical Report on C++ Library Extensions** provides one example of how such persistence facilities can be designed and implemented.

6. We want to have an interactive interface from our interpreters, hence a dictionary.

We believe this is not a central feature of the MATHLIB product. It is orthogonal functionality which would best be added by a system that uses the MATHLIB product. In fact, different languages or programming systems will require distinct mechanisms for "wrapping" the product. The MATHLIB product should be amenable to such "wrapping."

2.4 Critique of Analysis of Existing Software

The proposal includes a very brief analysis of existing software. It mentions GSL, GSL++ and CLHEP, making only a few comments about each. None of these are libraries which cover the scope of the proposed MATHLIB product, and so none could seriously be considered as fulfilling the need for a MATHLIB product. We believe the purpose of the comparison is to consider the *design* of each of the products; we undertake such a comparison below.

2.4.1 GSL

GSL is described as "very similar to a Fortran library"; its interface uses "arrays or special C-structs"; its "functions do not carry an internal state" and their level [of abstraction] is "very low." Let us consider these assertions in the context of some examples.

In the section "Object-Oriented API vs Procedural API," the proposal contains an example drawn for the GSL (and further critiqued below). That example shows the two interfaces for the gamma function. Of these interfaces, the first:

```
double gsl_sf_gamma(double x)
```

contains no arrays or structs. The second:

```
int gsl_sf_gamma_e(double x, gsl_sf_result* result)
```

makes use of the struct `gsl_sf_result`. This type is a C-struct that carries both the return value of the function and it associated uncertainty. While this

is an interesting feature, its use is peculiar to GSL. The first function shown corresponds to "normal" use, and conforms to a natural style that is adequate for this level of abstraction. The other mathematical special functions in GSL share the idiom of two interfaces -- one simple, and a second to deal with propagation of uncertainties.

Another reason that some of the GSL functions take arguments to pass algorithm "state" is to allow for thread safety. The GSL manual contains the following statements, explaining the use of such "workspace" variables.

> **Thread Safety**
>
> The library can be used in multi-threaded programs. All the functions are thread-safe, in the sense that they do not use static variables. Memory is always associated with objects and not with functions. For functions which use workspace objects as temporary storage the workspaces should be allocated on a per-thread basis. For functions which use table objects as read-only memory the tables can be used by multiple threads simultaneously. Table arguments are always declared const in function prototypes, to indicate that they may be safely accessed by different threads.

We believe that multithreaded programming will be an increasingly important consideration. From a technical perspective, algorithms that keep their own state often provide poor performance in a multithreaded environment, due to the need for locking. If such a library is designed to be thread-safe, the performance penalty is paid even by single-threaded programs. The passing of thread-specific data to algorithms is a standard technique to provide thread safety in a a high-performance system.

2.4.2 GSL++

We are unfamiliar with GSL++, and so can not comment on its quality of design or implementation.

2.4.3 CLHEP

CLHEP is described as "fulfilling also conditions 1 and 2, but not the essential conditions 3 and 4." As noted above, we disagree with the desire to have have the MATHLIB product favor any particular graphics environment. Furthermore, graphics is an orthogonal concern to be dealt with separately.

On the use of operator notation, we find this to be a factual error. CLHEP includes a multitude of operator definitions.

Finally, it is noted that "CLHEP is also a very small subset of what is required"; we agree, and will have more to say about coverage later in this document.

3 Library Organization

3.1 Summary of Proposal

The proposal (on page 2) contains a suggested organization of the library. This section of the proposal also discusses other issues, such as implementation of

some of the library functionality; we address some of these other issues elsewhere in this review.

The suggested organization is three groups of libraries; they are described as "shared libraries," presumably meaning dynamically linked libraries (as opposed to statically linked libraries). We believe that some users will wish to use static libraries, and some dynamic; both should be provided. We also note that the C++ Standard does not address the creation of dynamic libraries; as a result, the writing of portable dynamic libraries can be problematic -- extra care needs to be taken.

The suggested organization is three groups:

A. **Often used algorithms**, bundled into a single library. A *core library* is mentioned, into which this library is to be embedded.

B. **Classes for frequently used objects**; representative examples cited include:

   1. Physics vector classes,
   2. Random generators,
   3. Matrices,
   4. Parametric functions,
   5. Minimization classes.

   The proposal recommends that each of these subsets be put into its own library.

C. **Less frequently used algorithms**, bundled into a single "large" library.

3.2 Analysis of Proposal

We do not see any reason for the identification of these three "groups," nor do we see a description of what it means for several libraries to form a "group." We agree that multiple libraries are desirable, but believe the proposed "three logical groups" organization affords insufficient granularity.

> **We recommend** individual, cohesive libraries grouped by related functionality, for those libraries to form a hierarchy with as little coupling as possible, and for the libraries to have *no* circular dependencies at all.

In the following section (Candidate Library Organization), we present an alternate starting point for the organization of these libraries, representative of the level of granularity we recommend.

The proposal recommends the libraries contain "dictionaries"; presumably these are ROOT dictionaries. We believe that tight coupling of the proposed libraries to *any* external products entails disadvantages. These include:

1. Prerequisites introduce impediments for the use of the library, giving more things to install.
2. Prerequisites that are visible to the user increase the learning curve for use of the library.

3. Dependence on external products requires keeping link compatibility with the external product.
4. The dependent library must adapt whenever the external product changes, or risk slipping into the use of an unsupported version of the external product.

> **We recommend** that all aspects of the MATHLIB product be allowed to depend on a conforming implementation of Standard C++ and its library, free of vendor extensions.

> **We recommend** that individual libraries depend on external products only when those products directly supply functionality needed by that component library, and when the external products meet support and quality standards to be determined by CMS.

The proposal offers to "host the [product] as a component of the ROOT project." Embedding the MATHLIB product into an external product links this MATHLIB product to the release schedule of the product in which it is embedded. The CMS community would be better served by an independent product. It should be a relatively simple task, if the MATHLIB product is well-designed, to make it available through ROOT.

> **We recommend** against the subsumption of the MATHLIB product into ROOT. A design goal should be for the MATHLIB product to be easy to use in various environments.

The proposal recommends that the CVS structure should reflect the organization of the library structure. We agree with this.

> **We agree** with the proposal that the CVS structure reflect the organization of the library structure.

The proposal recommends creation of a "large" library for one of the "groups" of components in the product. When using a "large" dynamic library, the user pays the price of loading the entire library -- even if only a small amount of functionality from that library is required. This argues against production of libraries that are excessively "large."

> **CMS must be the arbiter** of what it considers "large," taking into account the expected computing resources of its users.

3.3 Candidate Library Organization

In the previous section, we recommended organizing the libraries constituting the MATHLIB product at a somewhat finer granularity than the proposal suggests. We here present, for CMS' consideration, a listing of possible component libraries as a candidate organization representative of the level of granularity we recommend.

This selection of functionality was derived from the tables in the proposal. CMS should review this list and amend it as necessary to suit its purposes optimally.

1. Polynomials -- evaluation, interpolation, binomial coefficients.
2. Nonlinear root finding
3. Special functions
4. Numerical integration -- quadrature rules, Monte Carlo
5. Maximization/minimization
6. Linear fitting
7. Nonlinear fitting
8. Linear algebra
9. Integration of differential equations
10. Interpolation, splines
11. Approximation of functions (Chebyshev series, *etc.*)
12. Random number generation
13. Probability distributions
14. Quantum mechanics (very limited)
15. Numerical differentiation
16. Descriptive statistics
17. Physical constants
18. Euclidean and Minkowskian geometry
19. Multidimensional interpolation, surface fitting
20. Quasirandom sequences

We stress that the above list is only preliminary. It is likely that comprehensive analysis will suggest changes of several kinds: some entries may be consolidated, some entries may be split, some entries may be eliminated, and new entries may appear.

By way of contrast, we view the following components as examples of candidate functionality unsuitable for inclusion by the MATHLIB product.

1. Data handling such as sorting and searching
2. Job time and date
3. HEP simulation, kinematics

The following recommendations encompass our reasoning regarding the above exclusions:

> **We recommend** that the MATHLIB product refrain from duplicating functionality already present in the C++ standard library.
>
> **We recommend** that the MATHLIB product avoid incorporating functionality that is inherently not mathematical in nature.
>
> **We recommend** that the MATHLIB product avoid incorporating functionality in the absence of widespread consensus on how such functionality should behave.

4 Scope and Coverage

4.1 Criteria for Inclusion in the Library

In order to assess any proposal with respect to its scope and coverage, it is necessary to have criteria that determine, in objective terms, the nature of the functionality desired, and against which proposed functionality can be evaluated. However, the proposal neither presents objective criteria of this nature nor does it refer to such criteria that may have been developed elsewhere.

We believe a discussion in CMS (and even in the wider HEP community) is necessary to discover these requirements. If such a discussion has already taken place, it should be directly referenced and addressed.

We are pleased to note that the proposal does include a detailed comparison of selected candidate functionality against the functionality provided by CERNLIB. This is an excellent approach for a prototype for the MATHLIB product's coverage, but more is needed.

> **We recommend** that CMS develop and document a list of requirements for the functionality desired of MATHLIB product, and that this list be sufficiently detailed as to guide the selection of library components.
>
> **We recommend** that a revised proposal be produced that directly addressees this list of requirements.

4.2 Resources Spanned

Sections 2 through 5 of the proposal present, in tabular form, summaries of four existing libraries that may serve as sources of desired functionality. Suggestions are made concerning which pieces of software may be taken (or, in the case of Fortran sources, used as a basis for re-writing).

As mentioned elsewhere in the review, we recommend a more finely-grained collection of libraries than is proposed. In this section, we recommend a possible allocation of the functionality suggested in the proposal to reflect this finer granularity.

As noted above, the coverage of the proposed library is already commendable; using the current CERNLIB as a benchmark for completeness is an excellent beginning. But CMS needs a clear statement of the functionality required of the MATHLIB product, in order to allow evaluation of the coverage. Additionally, more complete survey of freely available software would be valuable.

> **We recommend** additional review by CMS of the coverage of the proposed MATHLIB product. The appropriate groups within CMS should be asked to comment on the proposed scope and coverage of the library (distinct from being asked to comment on the proposed design).

Below, we present an analysis of the first two of the proposal's tables of functions, as examples of the sort of review we believe necessary for each table.

4.2.1 CERNLIB: "B - Elementary Functions"

This table presents only three functions out of the six that are listed in the CERNLIB documentation. No reason is given for the omission of the other functions. Since the missing functions are either not available or are defined differently in Standard C++, yet are a part of CERNLIB, their candidacy should be explored.

Time did not permit a detailed review of each table in the proposed document. Consistent with our recommendation above, we urge CMS to carry out such a detailed inspection in order to ensure that no important functionality has been omitted.

4.2.2 CERNLIB: "C - Equations and Special Functions"

This table contains a largely complete list of the functions in the referenced section of CERNLIB. We can identify, however, two different categories of functionality as are indicated by the title of the section:

1. root finding, solution of nonlinear systems of equations, and
2. evaluation of special mathematical functions.

Since these are orthogonal functionalities, we believe they should be presented in independent libraries.

Some of the functions mentioned are explicitly stated to deal with complex numbers (*e.g.* `CPOLYZ`, which calculates the zeros of a complex polynomial). We have seen few libraries which deal with complex numbers, and fewer which do so in a fashion natural to C++.

> **We recommend** that, if CMS needs functions of a complex variable, that such functionality be implemented using native C++, *e.g.*, using the C++ library classes `std::complex<double>`, `std::complex<float>`, and `std::complex<long double>`.

We would like to note that the C++ Standards Committee has accepted A Proposal to Add Mathematical Special Functions to the C++ Standard Library (version 3), which proposes the addition of a number of these functions to the C++ standard library.

> **We recommend** that the MATHLIB product provide a user interface to mathematical special functions.

See also Exploration of Selected Alternatives, below, for additional recommendations regarding design and implementation strategies for mathematical special functions as a part of the MATHLIB product.

5 Design

5.1 Criteria for Evaluation

The proposal discusses creation of a large number of individual libraries. There is no discussion of the design of each library. Rather, the expressed intention is to absorb whatever existing products are at hand, regardless of their design.

We disagree with this methodology. Once the scope of functionality for a given component library of the MATHLIB product has been identified, the design of that component needs to be proposed and reviewed. While some external sources may indeed provide well-designed component libraries ready for use, we believe it is far from certain that this will uniformly be true.

We further believe it premature to discuss the time and amount of effort required to implement a MATHLIB product before the remaining requirements are understood and a design is in hand.

5.2 Critique of Proposed Design

While the proposal does not address the design of the component libraries of the proposed MATHLIB product, it does present a design fragment pertaining to part of the *special functions* library. This example is presented in an attempt to prove the need for ubiquitous object-oriented design.

The proposal contrasts an "object-oriented API" to a "procedural API," and quickly settles on an object-oriented API for everything. We do not agree that object-oriented design is appropriate for all purposes. As we pointed out above, we believe there is more to a "true C++ library" than object-orientation.

5.2.1 A Case Study

Using the context of a gamma function in a special functions library, the proposal goes on to contrast an object-oriented API with a procedural API. However, the comparison is mixed, presenting function *prototypes* on the one hand but *user code* on the other hand.

Instead, let us compare, in both instances, only user code to evaluate the gamma function for a given argument:

Using a "procedural" paradigm (similar to GSL's, and as has been accepted into the Technical Report on C++ Library Extensions):

```
double y = gamma(x);          // x is a double
```

The object-based design (in what the proposal calls the "root style") is presented as:

```
TF1 gamma("gamma", TMath::Gamma, 0, 1);
double y = gamma.Eval(x);     // x is a double
```

We assume that `TMath` is a class, and `TMath::Gamma` is a static member function of that class. (The only other possibility is that `TMath` is a namespace, and `Gamma` is an element of that namespace.)

We find the procedural style to be clear and concise. Further, it seems to us to present an appropriate level of abstraction: the *implementation* of the gamma function is hidden, but not the fact that it is the gamma function that is being invoked.

In contrast, the root style seems error-prone. At the point of function invocation, the user names the *object* to which the function has been bound, rather than naming the actual function wanted. This extra level of indirection leaves an opportunity for mistakes to happen. This is illustrated by the following erroneous code:

```
TF1 gamma("beta", TMath::Zeta, 0, 1); // Zeta is the Riemann zeta function
// ...
// later, at a distant point in the code
double y = gamma.Eval(x);  // x is a double
```

Furthermore, the meaning of all but the second parameter of the `TF1` constructor is not obvious. If all we want to do is evaluate the function, are any of the other arguments necessary? Do the values of these arguments affect the evaluation of the gamma function?

Perhaps it would be more appropriate to contrast the procedural paradigm with direct calling of the static member function `TMath::Gamma`. An example of such use would be:

```
double y = TMath::Gamma(x);  // x is a double
```

This use is at best marginally object-oriented. The static member function is used in exactly the same manner as a free function. We can see no advantages to this design, when compared to a free function defined in a namespace.

We do, however, see a significant disadvantage to this design. The encapsulation provided by the class prevents user extension of the library: a user (such as CMS) can not add a new function which is on an equal footing with the functions provided by the library.

It is central to this analysis that the concept being abstracted is a mathematical function and that there is thus no need to allow two instances that differ based on internal state. In fact, there is no interesting internal state to query or manipulate. We therefore conclude that there is no need for an object orientation in modeling the evaluation of mathematical functions. Objects will be appropriate when there is an interesting internal state to query, to manipulate, and perhaps to persist, but none of these criteria seem to be met here.

It seems to be the goal of the proposal's class `TF1` to provide the user with certain predetermined abilities for each function. The examples given include graphing, calculation of the derivative at a point, and calculation of the integral over a range. We view these as orthogonal tasks; each seems best served by its own facility which may involve a class -- or possibly by a family of classes -- but which may involve another solution.

Conflating these facilities into a single class provides no obvious advantage. In fact, it has the disadvantage of making the library less extensible. It makes it harder, for example, for the user to add a new integration mechanism, on an equal footing with others in the library. Since the class `TF1` is closed (as is any class) to extension, user additions can not be put on an equal footing as the library-provided facilities.

Finally, the task of evaluating the mathematical function at a given point -- clearly the most common use -- does not require any of these extra facilities. The zero-overhead principle, central to much of C++ design, would therefore suggest that users ought not incur any cost of these unused facilities.

Extracted from the proposal's sole example, the above critique has studied only two classes, `TMath` and `TF1`: we find the design of each to be significantly flawed in the present context. To ensure the quality of the product, we believe that each component library of the MATHLIB product needs to have its design considered in comparable detail.

We recommend individual review of the design of each component library considered for inclusion in the MATHLIB product.

5.2.2 Exploration of Selected Alternatives

What are the different ways in which one might present mathematical functions in a library?

We present the following list to demonstrate that a wide variety of design possibilities is available for a C++ special functions library. Almost all of these design choices have some merits and some drawbacks. A clearly articulated design would require at least brief consideration of each, and a choice among the available trade-offs.

1. Using a *class* to encompass all the functions, one could implement each of the candidate functions via:

   - a non-virtual non-static member function,
   - a virtual non-static member function,
   - a non-static member function template,
   - a static member function (this is the choice found in the proposal), or
   - a static member function template.

2. Using a *class template* to encompass all the functions, one could implement each of the candidate functions via:

   - a non-virtual non-static member function,
   - a virtual non-static member function,
   - a non-static member function template,
   - a static member function, or
   - a static member function template.

3. One could use a class, or a class template, to represent each candidate function.

4. Finally, one could use free functions (encompassed by a namespace in order to avoid name collisions) to implement the candidate functions via:

   - an ordinary function (or an overloaded set thereof), or
   - a function template.

We have excluded, from the above list, any mention of inheritance and polymorphic behavior. This is because, in our judgment, the virtual function overhead would be unacceptable for many users of such classes.

As a further illustration of the sort of analysis required for each of the component libraries, we present the following analysis of the above options for the presentation of a special functions library.

1. The first group of solutions involves a single class (such at `TMath`) which presents the special functions in some manner. Of the five possibilities listed above, four have been implicitly rejected in the proposal. As to the remaining candidate (a static member function to implement each special function), we believe it to be a poor choice, largely for reasons presented during our case study in the previous subsection.

   Indeed, all solutions of this type share the same critical flaw: a class is closed to extension. As pointed out above, that means it is not possible for a user to add his own special function on an equal footing with the functions which are part of the library. Furthermore, it is not possible for the user to add a new data type and then have it handled by overloading existing functions in the library. We believe this flaw alone makes the entire family of solutions unsuitable.

   Are there advantages to a static member function over a free function? We can see none: it is invoked in the same manner, *etc.* Consider the code fragment:

   ```
   double y = mathlib_sf::gamma(x); // x is a double
   ```

   From this alone, it is indistinguishable whether `mathlib_sf` is a namespace containing a free function `gamma`, or `mathlib_sf` is a class having a static member function `gamma`. The only important difference is that a namespace is open, thus permitting library extensibility, while a class is closed, preventing extensibility.

2. The most significant difference distinguishing the second, class template-based, set of solutions from the first is that the second allows for support of user-defined data types: a user can specialize the member templates of interest for his new type. However, this solution has no benefit over a free function template in a namespace, for reasons similar to those above.

3. Although implicitly rejected by the proposal. the use of a class to represent each special function seems to be the most "object-oriented" solution of all. As pointed out earlier, special functions carry no interesting state, and hence there is no basis for distinguishing one instance of a given special function from another instance of the same special function. The other reasons cited above also apply. Hence, we agree this is not an appropriate approach in this context.

4. While we see no need to introduce function templates in this context, we do prefer the other solution from this last group: implement each of the special functions as a free function in a namespace. This allows for extensibility, since:

   - users can define new functions, and put them into the same namespace, and
   - users can overload exiting functions for new types of data.

This solution also (unsurprisingly) has the advantage of being the solution adopted by the C++ Standards Committee in working toward the next version of the C++ standard. It also has the advantage of easy implementation today, and easy migration to use of the functions in the future standard library when they become widely available.

We recommend that the special functions library be implemented as a collection of overloaded free functions in some defined namespace.

We recommend that the interfaces for those functions already proposed for inclusion in the future Standard be respected by the MATHLIB product's special functions library.

We recommend that functions added to the special functions library that are not part of the proposed future standard use a naming scheme and organization of signatures in a style similar to that proposed for the future standard.

We recommend that GSL be used as the underlying implementation for the first version of such a special functions library.

6 Conclusion

6.1 Summary

This document has presented a review of René Brun's paper of November 21, 2003, **Proposal for a C++ MATHLIB**. We have addressed the proposal paper from several overlapping viewpoints, including the organization, scope, and design of the proposed MATHLIB product.

In brief, we believe the underlying idea of the proposal, to create a MATHLIB product for use by the CMS and wider HEP communities, has significant potential. However, we also believe the proposal in its present form encompasses a number of flaws. We have attempted to identify these flaws and, in most cases, to provide guidance for their amelioration.

For convenience of the reader, the next section gathers all our recommendations, identifying the section from which each originated. Finally,

We recommend that the proposal be redrafted so as to address the weaknesses we have identified, and that the revised proposal be reviewed in turn.

6.2 Reprise of Recommendations

We collect here all the recommendations made earlier in this document, in the order in which they appear. Following each recommendation, we identify the section of this critique in which the recommendation originated.

1. We recommend that requirements for the MATHLIB product be formally stated. If such goals have been provided elsewhere, those goals should be referenced in this proposal.

2. We recommend the appropriate use of all the features of modern C++, in the appropriate contexts. The should be no *a priori* demand to use a single paradigm for all contexts. (Critique of Points in the Proposal)
3. We recommend individual, cohesive libraries grouped by related functionality, for those libraries to form a hierarchy with as little coupling as possible, and for the libraries to have *no* circular dependencies at all. (Analysis of Proposal)
4. We recommend that all aspects of the MATHLIB product be allowed to depend on a conforming implementation of Standard C++ and its library, free of vendor extensions. (Analysis of Proposal)
5. We recommend that individual libraries depend on external products only when those products directly supply functionality needed by that component library, and when the external products meet support and quality standards to be determined by CMS. (Analysis of Proposal)
6. We recommend against the subsumption of the MATHLIB product into ROOT. A design goal should be for the MATHLIB product to be easy to use in various environments. (Analysis of Proposal)
7. We agree with the proposal that the CVS structure reflect the organization of the library structure. (Analysis of Proposal)
8. CMS must be the arbiter of what it considers "large," taking into account the expected computing resources of its users. (Analysis of Proposal)
9. We recommend that the MATHLIB product refrain from duplicating functionality already present in the C++ standard library. (Candidate Library Organization)
10. We recommend that the MATHLIB product avoid incorporating functionality that is inherently not mathematical in nature. (Candidate Library Organization)
11. We recommend that the MATHLIB product avoid incorporating functionality in the absence of widespread consensus on how such functionality should behave. (Candidate Library Organization)
12. We recommend that CMS develop and document a list of requirements for the functionality desired of MATHLIB product, and that this list be sufficiently detailed as to guide the selection of library components. (Criteria for Inclusion in the Library)
13. We recommend that a revised proposal be produced that directly addressees this list of requirements. (Criteria for Inclusion in the Library)

14. We recommend additional review by CMS of the coverage of the proposed MATHLIB product. The appropriate groups within CMS should be asked to comment on the proposed scope and coverage of the library (distinct from being asked to comment on the proposed design). (Resources Spanned)
15. We recommend that CMS review the remaining unmentioned functions in CERNLIB, in each of the subsections listed in the proposal, to determine which functions are needed. (CERNLIB: "B - Elementary Functions")
16. We recommend that the organization of the MATHLIB product be unconstrained by CERNLIB's organization. and instead reflect its own desired cohesiveness. (CERNLIB: "B - Elementary Functions")
17. We recommend that, if CMS needs functions of a complex variable, that such functionality be implemented using native C++, *e.g.* using the C++ classes `std::complex<double>`, `std::complex<float>`, or `std::complex<long double>`. (CERNLIB: "C - Equations and Special Functions")
18. We recommend that the MATHLIB product provide a user interface to mathematical special functions. (CERNLIB: "C - Equations and Special Functions")
19. We recommend individual review of the design of each component library considered for inclusion in the MATHLIB product. (A Case Study)
20. We recommend that the special functions library be implemented as a collection of overloaded free functions in some defined namespace. (Exploration of Selected Alternatives)
21. We recommend that the interfaces for those functions already proposed for inclusion in the future Standard be respected by the MATHLIB product's special functions library. (Exploration of Selected Alternatives)
22. We recommend that functions added to the special functions library that are not part of the proposed future standard use a naming scheme and organization of signatures in a style similar to that proposed for the future standard. (Exploration of Selected Alternatives)
23. We recommend that GSL be used as the underlying implementation for the first version of such a special functions library. (Exploration of Selected Alternatives)
24. We recommend that the proposal be redrafted so as to address the weaknesses we have identified, and that the revised proposal be reviewed in turn. (Summary)

---

[1] A functional program is a single expression, which is executed by evaluating the expression. Anyone who has used a spreadsheet has experience of functional programming. In a spreadsheet, one specifies the value of each cell in terms of the values of other cells. The focus is on what is to be computed, not how it should be computed.

[2] New types and functions defined by clients should be treated on an equal basis with types and functions provided by the library.

[3] **Feedback from LHC Experiments on using CLHEP**; Moneta, Lorenzo, presented at the CLHEP workshop, 28 January 2003. Available as http://proj-clhep.web.cern.ch/proj-clhep/Workshop-2003/CLHEP_LHCfeedback.pdf